

AN Introduction to VHDL

Overview

Dinesh Sharma

Microelectronics Group, EE Department
IIT Bombay, Mumbai

January, 2014

Part I

VHDL Design Units

1 Design Units in VHDL

- entity
- Architecture
- Component
- Configuration
- Packages and Libraries

2 Object and Data Types

- Scalar data types
- Composite Data Types

An Introduction to VHDL

VHDL is a hardware description language which uses the syntax of ADA. Like any hardware description language, it is used for many purposes.

- For describing hardware.
- As a modeling language.
- For simulation of hardware.
- For early performance estimation of system architecture.
- For synthesis of hardware.
- For fault simulation, test and verification of designs.

etc.

VHDL versions

- 1987 First revised “Public” version.
- 1993 Probably the most widely used version.
Made syntax consistent, added xnor function etc.
- 2000 Minor revision: Introduced protected types.
- 2002 Minor revision: Rules with regard to buffer ports relaxed.
- 2008 Major revision: General clean up, introduced the use of external signals.

In this course, version 93 should be taken to mean 93 and later.

Design Elements in VHDL: ENTITY

The basic design element in VHDL is called an 'ENTITY'.

- An ENTITY represents a template for a hardware block.
- It describes just the outside view of a hardware module – namely its interface with other modules in terms of input and output signals.
- The hardware block can be the entire design, a part of it or indeed an entire “test bench”.
- A test bench includes the circuit being designed, blocks which apply test signals to it and those which monitor its output.
- The inner operation of the entity is described by an ARCHITECTURE associated with it.

ENTITY DECLARATION

The declaration of an ENTITY describes the signals which connect this hardware to the outside. These are called *port* signals. It also provides optional values of manifest constants. These are called generics.

VHDL 93 (and later)

```
entity name is  
    generic(list);  
    port(list);  
end entity name;
```

VHDL 87

```
entity name is  
    generic(list);  
    port(list);  
end name;
```

ENTITY EXAMPLE

VHDL 93 (and later)

```
entity flipflop is  
    generic (Tprop:delay_length);  
    port (clk, d: in bit; q: out bit);  
end entity flipflop;
```

VHDL 87

```
entity flipflop is  
    generic (Tprop: delay_length);  
    port (clk, d: in bit; q: out bit);  
end flipflop;
```

The entity declares port signals, their directions and data types.

These signals are used by an architecture associated with this entity.

Design Elements in VHDL: ARCHITECTURE

An ARCHITECTURE describes how an ENTITY operates. An ARCHITECTURE is always associated with an ENTITY.

There can be multiple ARCHITECTURES associated with an ENTITY.

An ARCHITECTURE can describe an entity in a structural style, behavioural style or mixed style.

The language provides constructs for describing components, their interconnects and composition (structural descriptions).

The language also includes signal assignments, sequential and concurrent statements for describing data and control flow, and for behavioural descriptions.

ARCHITECTURE Syntax

VHDL 93 (and later)

```
architecture name of entity-name  
is  
    (declarations)  
begin    (concurrent statements)  
end architecture name;
```

VHDL 87

```
architecture name of entity-name  
is  
    (declarations)  
begin    (concurrent statements)  
end name;
```

The architecture inherits the port signals from its entity. It must declare its internal signals. Concurrent statements constituting the architecture can be placed in any order.

ARCHITECTURE Example

VHDL 93 (and later)

```
architecture simple of dff is  
signal ...;  
begin  
...  
end architecture simple;
```

VHDL 87

```
architecture simple of dff is  
signal ...;  
begin  
...  
end simple;
```

Signals in an Architecture

- The architecture inherits the port signals from its entity. These signals have global scope inside **every** architecture associated with this entity.
- In addition to the port signals, an architecture may have internal signals. These must be declared in the declaration section.
- Internal signals declared here will have global scope inside **this particular architecture**.
- Local signals and variables may be declared inside internal blocks of the architecture.
- Signals are carried by wires, variables are used for array indices, loop counters, etc.

Design Elements in VHDL: COMPONENTS

- An ENTITY ↔ ARCHITECTURE pair actually describes a component **type**.
- In a design, we might use several **instances** of the same component **type**.
- Each instance of a component type may be distinguished by using a unique name.
- Thus, a component **instance** with a unique instance name is associated with a component **type**, which in turn is associated with an ENTITY ↔ ARCHITECTURE pair.
- This is like saying U1 (component instance) is a D Flip Flop (component *type*) which is associated with an entity DFF (which describes its pin diagram) using architecture LS7474 (which describes its inner operation).

Component Example

VHDL 93 (and later)

```
component name is  
    generic(list);  
    port(list);  
end component name;  
EXAMPLE:  
component flipflop is  
    generic (Tprop:delay_length);  
    port (clk, d: in bit; q: out bit);  
end component flipflop;
```

VHDL 87

```
component name  
    generic(list);  
    port(list);  
end component;  
EXAMPLE:  
component flipflop  
    generic (Tprop: delay_length);  
    port (clk, d: in bit; q: out bit);  
end component;
```

Design Elements in VHDL: Configuration

Structural Descriptions describe components and their interconnections.

A component is an instance of a component *type*. Each component type is associated with an ENTITY ↔ ARCHITECTURE pair.

The architecture used can itself contain other components - whose type will then be associated with other ENTITY ↔ ARCHITECTURE pairs.

A “**configuration**” describes linkages between component types and ENTITY ↔ ARCHITECTURE pairs. It specifies bindings for all components used in an architecture associated with an entity.

Design Elements in VHDL: Packages

Related declarations and design elements like subprograms and procedures can be placed in a "package" for re-use.

A package has a declarative part and an implementation part.

This is somewhat like entity and architecture for designs.

Objects in a package can be referred to by a `packagename.objectname` syntax.

A description can include a 'use' clause to incorporate the package in the design. Objects in the package then become visible to the description without having to use the dot reference as above.

Design Elements in VHDL: Libraries

Many design elements such as packages, definitions and entire entity architecture pairs can be placed in a library.

The description invokes the library by first declaring it:
For example, `Library IEEE;`

Objects in the Library can then be incorporated in the design by a 'use' clause.

For example, `Use IEEE.std_logic_1164.all`

In this example, IEEE is a library and std_logic_1164 is a package in the library.

Object and Data Types in VHDL

VHDL defines several types of **objects**. These include **constants**, **variables**, **signals** and **files**.

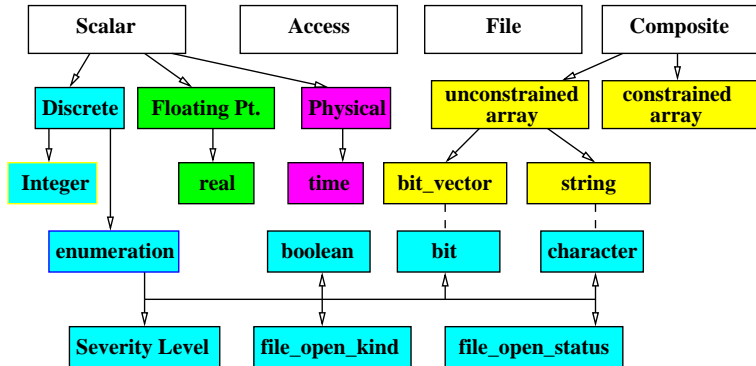
The types of values which can be assigned to these objects are called *data types*.

Same *data* types may be assigned to different *object* types. For example, a **constant**, a **variable** and a **signal** can all have values which are of *data type* **BIT**.

Declarations of objects include their *object* type as well as the *data* type of values that they can acquire.

For example **signal** Enable: **BIT**;

Data Types



Enumeration Type

VHDL *enumeration* types allow us to define a set of values that an object of this type can acquire. For example, we can define a data type by the following declaration:

```
type instr is (add, sub, adc, sbb, rotl, rotr);
```

Now a variable or a signal defined to be of type instr can only be assigned values enumerated above – that is: add, sub, adc, sbb, rotl and rotr.

In actual implementation, these values may be mapped to a 3 bit value. However, an attempt to assign, say, “010” to a variable of type instr will result in an error. Only the enumerated values can be assigned to a variable of this type.

Pre-defined Enumeration Types

A few enumeration types are pre-defined in the language.

These are:

type bit **is** ('0', '1');

type Boolean **is** (false, true);

type severity_level **is** (note, warning, error, failure);

type file_open_kind **is** (read_mode, write_mode, append_mode);

type file_open_status **is**
(open_ok, status_error, name_error, mode_error);

In addition to these, the character type enumerates all the ASCII characters.

Types and SubTypes

A signal type defined in the IEEE Library is `std_logic`. This is a signal which can take one of 9 possible values. It is defined by:

```
type std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

A subtype of this kind of signal can be defined, which can take the four values 'X', '0', '1', and 'Z' only.

This can be defined to be a subtype of `std_logic`

```
subtype fourval_logic is std_logic range 'X' to 'Z';
```

Similarly, we may want to constrain some integers to a limited range of values. This can be done by defining a new type:

```
subtype bitnum is integer range 31 downto 0;
```

Physical Types

Objects which are declared to be of Physical type, carry a value as well as a unit. These are used to represent physical quantities such as time, resistance and capacitance.

The Physical type defines a basic unit for the quantity and may define other units which are multiples of this unit.

Time is the only Physical type, which is pre-defined in the language. The user may define other Physical types.

Pre-defined Physical Type: Time

```
type time is range 0 to ...  
  units  
    fs;  
    ps = 1000 fs;  
    ns = 1000 ps;  
    us = 1000 ns;  
    ms = 1000 us;  
    sec = 1000 ms;  
    min = 60 sec;  
    hr = 60 min;  
  end units time;
```

The user may define other physical types as required.

User Defined Physical Types

As an example of user defined Physical types, we can define the resistance type.

```
type resistance is range 0 to 1E9  
    units  
        ohm;  
        kohm = 1000 ohm;  
        Mohm = 1000 kohm;  
    end units resistance;
```


Composite Data Types

Composite data types are collections of scalar types.

VHDL recognizes records and arrays as composite data types.

Records are like structures in C.

Arrays are indexed collections of scalar types. The index must be a discrete scalar type.

Arrays may be one-dimensional or multi-dimensional.

Arrays

Arrays can be constrained or unconstrained.

- In constrained arrays, the type definition itself places bounds on index values. For example:

type byte **is array** (7 downto 0) **of** bit;

type rotmatrix **is array** (1 to 3, 1 to 3) **of** real;

- In unconstrained arrays, no bounds are placed on index values. Bounds are established at the time of declaration.

type bus **is array** (natural range <>) **of** bit;

The declaration could be:

signal addr_bus: bus(15 downto 0);

signal data_bus: bus(7 downto 0);

Built in Array types

VHDL defines two built in types of arrays. These are: bit_vectors and strings. Both are unconstrained.

type bit_vector **is array** (natural range <>) **of** bit;

type string **is array** (positive range <>) **of** character;

As a result we can directly declare:

variable message: string(1 to 20);

signal Areg: bit_vector(7 downto 0);

Records

While an array is a collection of the same type of objects, a record can hold components of different types and sizes.

This is like a struct in C.

The syntax of a record declaration contains a semicolon separated list of fields, each field having the format name, ..., name : type or name, ..., name : subtype

For example:

type resource **is record**

(P_reg, Q_reg : bit_vector(7 downto 0); Enable: bit)

end record resource;

Part II

Structural Description in VHDL

3 Structural Description

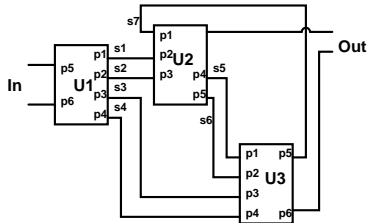
- Component Declarations
- Component Instantiation
- Configuration
- Repetition Grammar

Structural Style

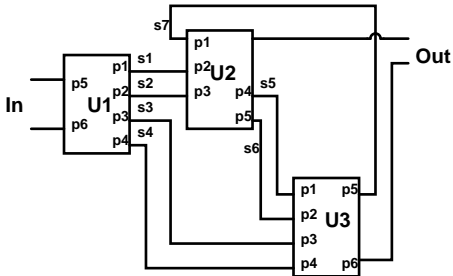
Structural style describes a design in terms of components and their interconnections.

Each component declares its ports and the type and direction of signals that it expects through them

How can we describe interconnections between components?



Describing Interconnect



For each internal interconnect, we define an internal signal.

When instantiating a component, we *map* its ports to specific internal signals.

- For example, in the circuit above, At the time of instantiating U1, we map its pin p2 to signal s2.
- Similarly, when instantiating U2, we map its pin p3 to s2.
- This connects p2 of U1 to s2 and through s2 to pin p3 of U2.

Structural Architecture

A purely structural architecture for an entity will consist of

- 1 Component declarations: to associate component *types* with their port lists.
- 2 Signal Declarations: to declare the signals used.
- 3 Component Instantiations: to place component instances and to portmap their ports to signals. Signals can be internal or port signals declared by the ENTITY.
- 4 Configurations: to bind component types to ENTITY ↔ ARCHITECTURE pairs.
- 5 Repetition grammar: for describing multiple instances of the same component type – for example, memory cells or bus buffers.

Component Declarations

VHDL 93

component name **is**
 generic(list);
 port(list);
end component name;

EXAMPLE:

```
component flipflop is
    generic (Tprop:delay_length);
    port (clk, d: in bit; q: out bit);
end component flipflop;
```

VHDL 87

component name
 generic(list);
 port(list);
end component;

EXAMPLE:

```
component flipflop
    generic (Tprop: delay_length);
    port (clk, d: in bit; q: out bit);
end component;
```

Component Instantiation

VHDL-93: Direct Instantiation

VHDL-93 allows direct instantiation of ENTITY ↔ ARCHITECTURE pairs without having to go through a component *type* declaration first.

Instance-name: **entity** entity-name (architecture-name)
 generic map(list)
 port map(list);

This form is convenient, but does not have the flexibility of associating alternative ENTITY ↔ ARCHITECTURE pairs with a component.

VHDL-87 does not allow direct instantiation.

Component Instantiation

VHDL-93: Normal Instantiation

Instance-name: **component** component-type-name
 generic map(list)
 port map(list);

The association here is with a previously declared component type. The type will be bound to an ENTITY↔ ARCHITECTURE pair using an inline configuration statement or a configuration construct.

Component Instantiation

VHDL-87

The keyword **component** is not used in VHDL-87. This is because direct instantiations are not allowed and therefore the binding is *always* to a component.

Instance-name: component-type-name
 generic map(list)
 port map(list);

The association is with a previously declared component type. The type will be bound to an ENTITY ↔ ARCHITECTURE pair using an inline configuration statement or construct.

Inline Configuration

The association between component types and ENTITY↔ARCHITECTURE pairs can be made inline with a *use* clause.

for all: component-name
 use entity entity-name(architecture-name);

Instead of saying **for all**, we can specify a list of selected instances of this component type to which this binding will apply.

instance-name-list: component-name
 use entity entity-name(architecture-name);

The keyword OTHERS

If we use the keyword **others** instead of a list of instance names, it refers to all component instances of this component-name which have not yet figured in a name-list.

In VHDL, the keyword **others** is used in different contexts involving lists.

If some members of the list have been specified, then **others** refers to the remaining members. (If none was specified, it is equivalent to **all**.)

Hierarchical Configuration

When we associate a *component type* with a previously defined ENTITY \leftrightarrow ARCHITECTURE pair, the chosen architecture could itself contain other components - and these components in turn would be associated with other ENTITY \leftrightarrow ARCHITECTURE pairs.

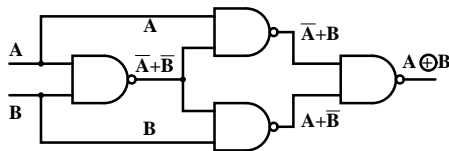
This hierarchical association can be described by a standalone design unit called a **configuration**.

Hierarchical Configuration

VHDL contains fairly complex configuration statements. A simplified construct is introduced here:

```
configuration config-name of entity-name is  
  for architecture-name  
    for component-instance-namelist: component-type-name  
      use entity entity-name(architecture-name);  
    end for  
  end for  
end configuration config-name;
```


Structural description: Example



- Let us choose the xor gate shown on the left as an example for structural description.
- It uses four instances of a single type of component: two-input NAND.
- We shall describe the NAND gate first.

The work library

- In VHDL, as we describe entities and architectures, these are compiled into a special library called **WORK**.
- This library is always included and does not have to be declared.
- In some sense, the WORK library represent the current state of development of the project for designing something.

Definition of NAND

Entity nand2 is

port (in1, in2: in bit; p: out bit);

end entity nand2;

We do not use any generic for this simple example.

Architecture trivial of nand2 is

p <= not (in1 and in2);

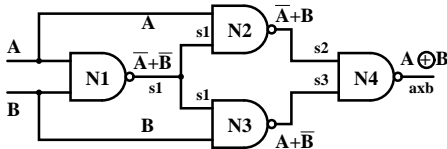
end Architecture trivial;

‘not’ and ‘and’ are inbuilt logical functions.

(Actually so is nand – but we are trying to be cute!)

Now that we have this entity-architecture pair, we can use it to build our xor gate.

XOR Gate example



USE WORK.ALL

Entity xor is

port(a,b: in bit; axb: out bit);

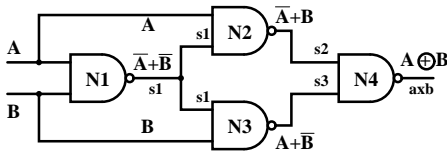
End Entity xor;

Architecture simple of xor is
 component NAND2in IS port(a,b:
 in bit; axb: out bit);

For all NAND2in: use Entity
 NAND2(Trivial);

signal s1,s2,s3: bit;

XOR Architecture body



```

begin
N1: component NAND2in
portmap(a, b, s1);
N2: component NAND2in
portmap(a, s1, s2);
N3: component NAND2in
portmap(b, s1, s3);
N4: component NAND2in
portmap(s2, s3, axb);
end Architecture simple;
  
```

Repetition Grammar

We frequently use a large number of identical components of the same *type*. (For example memory cells or bus drivers). It is tedious to instantiate and configure each one of them individually.

VHDL provides a way to place a collection of instances of a component *type* at one go using the **generate** statement.

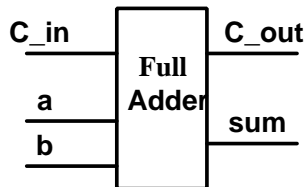
GENERATE Statement

The generate statement contains a for loop which takes effect during the **circuit elaboration** step. This can be used to repeat instantiation constructs. We illustrate this statement with an example:

```
groupname: for index in 0 to width-1 generate  
begin  
    some-name: component outbuf  
        portmap (...);  
end generate groupname;
```

The defined index in the “for” construct has local scope and can be used to pick specific signals from an array in portmap statements.

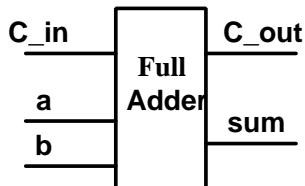
Example: Full adder



Entity FullAdder is
Port(a,b, C_in: in bit; sum, C_out: out bit);
End Entity FullAdder;

C_out and sum represent the more significant and less significant bits of $a+b+C_{in}$.

Example: Full adder



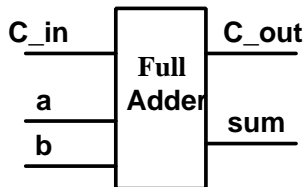
Entity FullAdder is
Port(a,b, C_in: in bit; sum, C_out: out bit);
End Entity FullAdder;

C_out and sum represent the more significant and less significant bits of $a+b+C_{in}$.

Suppose this is too difficult for the likes of us to figure out



Example: Full adder



Entity FullAdder is
Port(a,b, C_in: in bit; sum, C_out: out bit);
End Entity FullAdder;

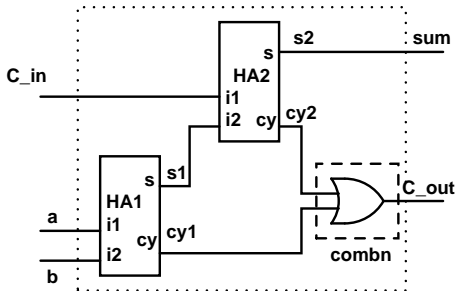
C_out and sum represent the more significant and less significant bits of $a+b+C_{in}$.

Suppose this is too difficult for the likes of us to figure out



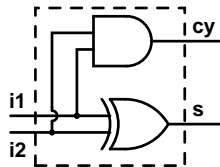
We would like to decompose the circuit into blocks which handle two bits at a time.

Decomposition of Full Adder



The combiner just combines the carries from the two half adders. (Just an OR Gate will do it.)

Half Adder



Each half adder represents the sum and carry of just two bits.

Carry occurs only if both bits are 1. Sum is zero if both bits are zero or both are one.

so, $s = a \text{ xor } b$, $cy = a \text{ and } b$.

Description of full Adder

```
Entity HalfAdder is  
port(in1, in2: in bit; s, cy: out bit);  
End Entity HalfAdder;
```

```
Architecture trivial of HalfAdder is  
begin  
    s <= a xor b;  
    cy <= a and b;  
end Architecture trivial;
```

Architecture simple of FullAdder is
Component HalfAdder is

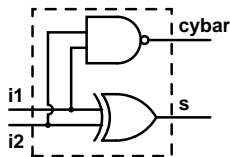
```
    port(a, b: in bit; s, cy: out bit);  
End Component HalfAdder;  
signal s1, cy1, cy2: bit;  
begin  
    HA1: Component HalfAdder  
        portmap(a,b,s1,cy1)  
    HA2: Component HalfAdder  
        portmap(s1,C_in,sum,cy2)  
    Cmbn: Component OR2in  
        portmap(cy1, cy2, C_out)  
end Architecture simple;
```

The half adder

Carry from the half adder is an AND gate, and the combiner is an OR.

But Gates without inversion are slow. So we bring out $\overline{\text{carry}}$ rather than carry, using a NAND gate.

Half Adder



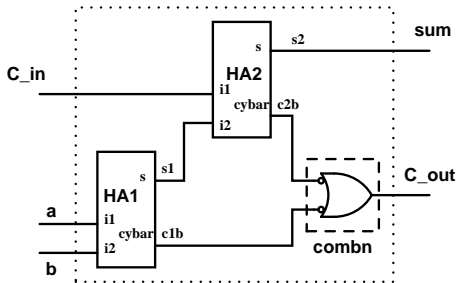
Entity HalfAdder is
 port(in1, in2: in bit; s, cybar: out bit);
 End Entity HalfAdder;
 Architecture better of HalfAdder is
 begin

```
s <= a xor b;
cybar <= a nand b;
```

end Architecture better;

The combiner should now be an OR of negative true signals, which is just a NAND

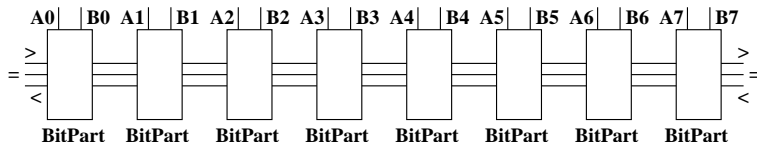
Efficient Full Adder



Architecture better of FullAdder is
 Component HalfAdder is
 port(a, b: in bit; s, cybar: out bit);
 End Component HalfAdder;
 signal s1, c1b, c2b: bit;
 begin
 HA1: Component HalfAdder
 portmap(a,b,s1,c1b);
 HA2: Component HalfAdder
 portmap(s1,C_in,sum,c2b);
 Cmbn: Component NAND2in
 portmap(c1b, c2b, C_out);
 end Architecture better;

Decomposition of Byte Comparator

The byte comparator is difficult to design directly.
We can break up the design into bit comparators
with cascading inputs `gt_in`, `eq_in` and `lt_in`;
and cascading outputs `gt_out`, `eq_out` and `lt_out`.



Notice that the most significant bit is compared closest to the output.

Composing the Byte comparator

Architecture compose of Byte_Compar IS

```
    COMPONENT BitPart IS
```

```
        Port(a, b: IN std_ulogic;
```

```
              gt_in, eq_in, lt_in: IN std_ulogic;
```

```
              gt_out, eq_out, lt_out: OUT std_ulogic);
```

```
    END COMPONENT BitPart;
```

```
    FOR ALL: BitPart
```

```
        USE ENTITY Bit_Compar(behavior);
```

```
    TYPE Connect IS ARRAY (1 TO 3, 0 TO 6) OF std_ulogic);
```

```
    Signal Cascade: Connect;
```


Composing the Byte comparator

```
BEGIN
  FOR I in 0 TO 7 GENERATE
    First: IF I = 0 GENERATE
      COMPONENT BitPart
        PORTMAP
          (gt_in, eq_in, lt_in,
           a(I), b(I),
           Connect(1, I), Connect(2, I), Connect(3, I));
      END GENERATE;
```

Composing the Byte comparator

```
Last: IF I = 7 GENERATE
      COMPONENT BitPart
        PORTMAP
          (Connect(1, I-1), Connect(2,I-1), Connect(3,I-1));
          a(I), b(I),
          gt_out, eq_out, lt_out)
      END GENERATE;
```

Composing the Byte comparator

```
Mid: IF (I > 0) AND (I < 7) GENERATE
      COMPONENT BitPart
        PORTMAP
          (Connect(1, I-1), Connect(2, I-1), Connect(3, I-1));
          a(I), b(I),
          Connect(1, I), Connect(2, I), Connect(3, I));
      END GENERATE;
END GENERATE;
END Architecture Compose;
```

The bit comparator

Once we have decomposed the byte comparator as above, we need to design the bit comparator.

- The bit comparators receive a pair of bits to compare.
- If $A > B$, i.e. $A=1$ and $B=0$; it makes the output `gt_out` TRUE and makes the other outputs FALSE.
- If $A < B$, i.e. $A=0$ and $B=1$; it makes the output `lt_out` TRUE and makes the other outputs FALSE.
- IF A and B are equal, it copies its cascading inputs (`gt_in`, `eq_in`, `lt_in`) to its outputs (`gt_out`, `eq_out`, `lt_out`);

The bit comparator

```
Library IEEE;  
USE IEEE.std_logic_1164.ALL;  
  
Entity Bit_Compar is  
    Port(a, b: IN std_ulogic;  
          gt_in, eq_in, lt_in: IN std_ulogic;  
          gt_out, eq_out, lt_out: OUT std_ulogic);  
End Entity Bit_Compar;
```

Behavioural Architecture of Bit Comparator

Architecture behave Of Bit_Compar is

BEGIN

P1: PROCESS(a, b, gt_in, eq_in, lt_in)

BEGIN

IF (a = '1' AND b = '0') THEN

gt_out <= '1'; eq_out <= '0'; lt_out <= '0';

ELSIF (a = '0' AND b = '1') THEN

gt_out <= '0'; eq_out <= '0'; lt_out <= '1';

ELSE gt_out <= gt_in; eq_out <= eq_in; lt_out <= lt_in;

END IF;

END PROCESS P1;

END Architecture behave;

Structural Description of Bit Comparator

We can write Karnaugh Maps for the three outputs easily:

gt_out				
ab → gt_in ↓	00	01	11	10
0				✓
1	✓		✓	✓

lt_out				
ab → lt_in ↓	00	01	11	10
0		✓		
1	✓	✓	✓	

eq_out				
ab → eq_in ↓	00	01	11	10
0				
1	✓		✓	

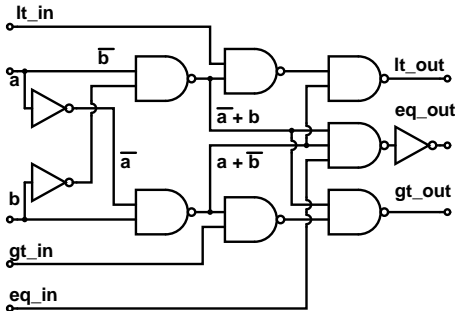
This gives:

$$gt_out = a \cdot \bar{b} + gt_in \cdot (a + \bar{b})$$

$$lt_out = \bar{a} \cdot b + lt_in \cdot (\bar{a} + b)$$

$$eq_out = eq_in \cdot (a \cdot b + \bar{a} \cdot \bar{b})$$

Final Design of bit comparator



- This design can be described structurally in terms of basic gates.
- The design uses only inverting gates. It can be implemented directly on a chip.

Structural Description of Bit Comparator

Architecture struct Of Bit_Compar is

Component Inv IS

PORT(In1: IN std_ulogic; op1: OUT std_ulogic);

END COMPONENT Inv;

FOR ALL: Inv USE ENTITY Inverter(behav);

Component Nand2 IS

PORT(In1, In2: IN std_ulogic; op1: OUT std_ulogic);

END COMPONENT Nand2;

FOR ALL: Nand2 USE ENTITY Nand2(behav);

Component Nand3 IS

PORT(In1, In2, In3: IN std_ulogic; op1: OUT std_ulogic);

END COMPONENT Nand3;

FOR ALL: Nand3 USE ENTITY Nand3(behav);

Structural Architecture of Bit Comparator

```
Inv1:    Inv PORTMAP(A, Abar);
Inv2:    Inv PORTMAP(B, Bbar);
N1:      Nand2 PORTMAP(A, Bbar, BplusAbar);
N2:      Nand2 PORTMAP(B, Abar, AplusBbar);
N3:      Nand2 PORTMAP(lt_in, BplusAbar, s1);
N4:      Nand2 PORTMAP(gt_in, AplusBbar, s2);
N5:      Nand2 PORTMAP(s1, AplusBbar, lt_out);
N6:      Nand2 PORTMAP(s2, BplusAbar, gt_out);
N7:      Nand3 PORTMAP(AplusBbar, BplusAbar, Eq_in, Eqbar);
Inv3:    Inv PORTMAP(Eqbar, Eq_out);
```

Inline configuration

The configuration of a component can be declared “inline” in an architecture.

Architecture compose of Byte_Compar IS

```
    COMPONENT BitPart IS
```

```
        Port(a, b: IN std_ulogic;
```

```
              gt_in, eq_in, lt_in: IN std_ulogic;
```

```
              gt_out, eq_out, lt_out: OUT std_ulogic);
```

```
    END COMPONENT BitPart;
```

```
    FOR ALL: BitPart
```

```
        USE ENTITY Bit_Compar(behave);
```

```
    TYPE Connect IS ARRAY (1 TO 3, 0 TO 6) OF std_ulogic);
```

```
    Signal Cascade: Connect;
```

All components of type BitPart have been configured to use the



Stand alone configuration

We can write a separate configuration description outside the architecture using the configuration. The syntax is:

```
CONFIGURATION configname OF entityname IS
    FOR architecture_name
        FOR instance_name | OTHERS | ALL : component_name
            USE ENTITY sub_entity_name(sub_architecture_name)
            ...
        END FOR;
    END FOR;
END [CONFIGURATION] [configname];
```

Different instances of the same component type can be bound to different entity-architecture pairs.

Hierarchical configuration

- The architecture being configured may contains components which are bound to architectures containing other components.
- This requires hierarchical configuration.
- Instead of binding component instances to entity-architecture pairs directly, we bind these to other configurations.
- These other configurations associate the component with an entity-architecture pair and cofigure the lower level components.

Hierarchical configuration

The syntax used for hierarchical configuration is:

```
CONFIGURATION configname OF entityname IS
    FOR architecture_name
        FOR instance_name | OTHERS | ALL : component_name
            USE CONFIGURATION subconfig_name;
        ...
    END FOR;
END FOR;
END [CONFIGURATION] [configname];
```

Subconfig_name will associate the component with an entity-architecture pair and will configure lower level components in the hierarchy.

Example of Hierarchical Configuration

Consider the byte comparator example:

Architecture compose of Byte_Compar IS

```
    COMPONENT BitPart IS
```

```
        Port(a, b: IN std_ulogic;
```

```
            gt_in, eq_in, lt_in: IN std_ulogic;
```

```
            gt_out, eq_out, lt_out: OUT std_ulogic);
```

```
    END COMPONENT BitPart;
```

```
    FOR ALL: BitPart
```

```
        USE CONFIGURATION BitPartConfig;
```

```
    ...
```

Part IV

Behavioural Description Using VHDL

- 4 Behavioural Description
 - Concurrent Statements
 - Processes
 - Sequential Statements
- 5 Attributes
 - Array attributes
 - Type Attributes
 - Signal attributes

Behavioural Style

Behavioural style describes a design in terms of its behaviour, and not in terms of a netlist of components.

We describe behaviour through “if-then-else” type of constructs, loops, sequential and concurrent assignment statements.

Statements like “if-then-else” are inherently sequential. These must therefore occur only inside sequential bodies like processes.

A concurrent assignment statement may be considered as a shorthand for a very simple process.

Specifying a waveform

A waveform is described by a comma separated list of values and optionally, delays. For example, we may assign a waveform by a statement like

```
indata <= '0', '1' AFTER 20 NS, '0' AFTER 50 NS;
```

The values at different times are treated as **transport** delays and are all inserted in the time ordered queue without wiping out earlier values.

(This is the only context where delays are transport by default).
Single value assignments use inertial delay by default.

Concurrent Assignment

A concurrent assignment can be made conditionally by using 'when' clauses.

```
name <= [delay-mechanism]  
    waveform when Boolean-expression else  
    waveform when Boolean-expression;
```

The assignment is made from the first waveform where the Boolean expression evaluates to TRUE.

Concurrent Assignment

The assignment can also be made on a selective basis, based on the value of some expression:

with expression **select**
name <= [delay-mechanism]
 waveform **when** choices,
 waveform **when** choices;

If the expression evaluates to one of the specified choices, the corresponding assignment is made.

Assignment to an aggregate

Assignments can be made to a collection of signals simultaneously. For example let `vec` be defined as `bit_vector(2 downto 0)`

```
vec <= ("000") - - 000 : string
```

```
vec <= ('0','0','1') - - 001 : positional
```

```
vec <= (1=>'1', others => '0') - - 010 : named, partial
```

```
vec <= ('1', others => '0') - - 100 : positional, partial
```

```
vec <= (2|0=>'1', others => '0') - - 101 : partial
```

```
vec <= (others => '1') - - 111
```

Processes

Sequential constructs need to be placed inside a process. A process uses the syntax:

```
[ process-label: ] process [(sensitivity-list)] [is]  
[declarations]  
begin  
    [sequential statements]  
end process [process-label];
```

Sequential statements include “if” constructs, case statements, looping constructs, assertions, wait statements etc.

Process with Sensitivity list

Every process is like an endless loop. Therefore, it requires an explicit or implicit suspend statement.

If a sensitivity list is given with the process statement, the process automatically suspends when it reaches its end.

It restarts from the beginning when any of the signals in its sensitivity list has an event.

This process has a *static* sensitivity and an *implicit* suspend statement.

Wait statements

A process without a sensitivity list requires explicit suspend statements. These are provided by wait statements. These can be of the form:

wait for waiting-time;
wait on signal-list;
wait until waiting-condition;
wait for 0 some-time-unit;
wait;

wait for 0 ns causes the process to suspend till the next delta. The last form (bare wait statement) suspends the process for ever.

Dynamic sensitivity

Processes without a sensitivity list and multiple wait statements have a *dynamic* sensitivity. This is because these processes are sensitive to different events at different times.

One cannot mix static and and dynamic sensitivity
Thus, a process with a sensitivity list cannot use wait statements.

This is because once the process is suspended, it is possible to have an event on a signal in the sensitivity list *simultaneously* with the condition for resumption after wait being fulfilled.

This would leave the process undecided on *where* to resume from.

IF statements

if statements are similar to their counterparts in programming languages. The syntax is:

```
[ if-label: ] if Boolean-expression then  
    sequential statements  
  [ elsif Boolean-expression then  
    sequential statements ]  
  [ elsif ... ]  
  [ else sequential statements ]  
end if [ if-label ];
```

CASE statements

A case statement acts like a multiplexer.

The syntax is:

```
[ case-label:] case expression is  
    when choices = >  
        sequential-statements  
    [ when ... ]  
end case [ case-label ];
```

CASE Choices

Choices can be specified in CASE statements as vertical bar separated lists of expressions, discrete ranges or the keyword **others**. For example:

case opcode **is**

load | store | add | subtract = >

...

Loop Statements

There are several different forms of the loop statement. The simplest is the endless loop:

```
[ loop-label: ] loop  
[ loop-label: ] loop  
    sequential statements  
end loop [ loop-label ];
```

This constitutes an endless loop.

It is assumed that it will have an exit statement or a wait statement inside to suspend operation.

Exiting a Loop

The exit statement has the syntax:

[label:] **exit** [loop-label] [**when** Boolean expression]

The loop label allows one to exit several levels of nested loops.

We can also skip to the end of a loop by using the **next** statement. This works like “continue” in C.

NEXT Statement

```
[ label: ] next [ loop-label ] [ when Boolean expression ]
```

The **next** statement skips the statements of the loop and immediately starts the next iteration of the specified loop.

The loop label allows one to skip through several levels of nested loops.

WHILE Loops

VHDL also has a **while** loop.

```
[ loop-label: ]  
while Boolean-expression loop  
    sequential statements  
end loop [ loop-label ];
```

The loop continues as long as the Boolean expression is TRUE.

For Loops

VHDL also provides a **for** loop.

```
[ loop-label: ]  
for identifier in discrete-range loop  
    sequential statements  
end loop [ loop-label ];
```

The discrete range can be of the form
expression **to** | **downto** expression

The identifier is initialized to the left limit of the range and takes on successive values in the discrete range till it exceeds the right limit.

Assertions and Reports

The assert statement takes the form

```
[ label: ] assert Boolean expression  
    [ report expression ] [ severity expression ];
```

If the Boolean expression is TRUE, no action is taken.

If it is FALSE, an assertion violation is said to have occurred.

The simulators then outputs the **report** expression.

Subsequent operation depends on the severity clause.

Severity Clause in Assertions

Assert statements are used for debugging and documentation. The severity clause decides what happens when an assertion failure occurs.

Severity is an enumerated type which is predefined to take any of the values:

note, warning, error, failure

Depending on the severity value, simulation continues or is aborted.

Severity values

Note is simply to generate an output when an assertion violation occurs.

Warning is useful when the validity of the simulation may be in doubt, but we would like to issue a warning and continue anyway.

Error is used when an unexpected value is encountered.

Failure is the most severe violation and is used when some inconsistency is detected.

Assertions defaults

[label:] **assert** Boolean expression
[**report** expression] [**severity** expression];

If the optional report clause is missing in the assert statement, the default report message is “Assertion Violation”.

If the severity clause is omitted, the default value is ‘error’.

Most simulators allow the user to set a severity threshold, beyond which the simulation is aborted on an assertion violation. It is common to continue on note and warning and to abort on error and failure.

In VHDL-93, the report clause can be used by itself as a statement to output useful messages.

Attributes

VHDL provides built in functions which return usefult attributes of the objects that they operate on.

Attribute functions may provide attributes of

- Arrays
- Types
- Signals
- Entities

Attributes are invoked as `name'attrib_name`.
The single quote is read as “tick”

Array Attributes

Array attributes interrogate the property of arrays. Consider the declaration:

```
TYPE regfile IS ARRAY(0 To 3, 7 Downto 0) OF BIT;
```

Then we can use the following attributes:

'LEFT :

```
regfile'LEFT(2) = 7
```

'RIGHT:

```
regfile'RIGHT(1) = 3
```

'HIGH:

```
regfile'HIGH(2) = 7
```

'LOW:

```
regfile'LOW(1) = 0
```

'RANGE:

```
regfile'RANGE(1)= 0 TO 3
```

'REVERSE_RANGE:

```
regfile'REVERSE_RANGE(1) = 3  
DOWNT0 0
```

'LENGTH: regfile'LENGTH(1) = 4

'ASCENDING:

```
regfile'ASCENDING(1) = TRUE
```

Type Attributes

Type attributes apply only to scalar types. Consider the declarations:

```
TYPE nineval IS('U', 'X', '0', '1', 'Z', 'L', 'H', 'W', '-')
```

```
SUBTYPE fourval IS nineval RANGE 'X' to 'Z'
```

Then, fourval'BASE = nineval

Attributes LEFT, RIGHT, HIGH and LOW are defined for TYPES also. When applied to a TYPE, these return the corresponding values as defined for the type. For example,

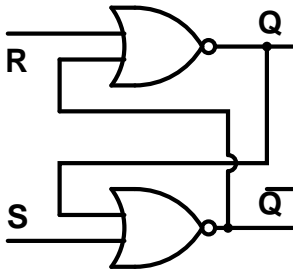
```
nineval'LEFT = 'U', fourval'LEFT = 'X'
```

```
POSITIVE'LOW = 1
```


Signal Attributes

Name	Example	Return type	Value type
'DELAYED	s'DELAYED	Signal	same as s
'STABLE	s'STABLE(5ns)	Signal	Boolean
'EVENT	s'EVENT	Value	Boolean
'QUIET	s'QUIET(3ns)	Signal	Boolean
'TRANSACTION	s'TRANSACTION	Signal	BIT
'DRIVING	s'DRIVING	Value	Boolean
'DRIVING_VALUE	s'DRIVING_VALUE	Value	same as s

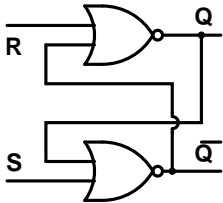
Case of RS Latch



```
Entity RS_Latch is  
Port(R,S: IN BIT; Q, Qbar: OUT BIT);  
End Entity RS_Latch;  
Architecture trouble of RS_Latch is  
Begin  
  Q <= R NOR Qbar;  
  Qbar <= S NOR Q;  
End Architecture trouble;
```

This will run into trouble as Q and Qbar are declared to be outputs and cannot be used on the RHS expression of an assignment.

RS Latch



We have several choices:

Declare Q and Qbar to be inout.

This is not safe as this will allow outside circuitry to drive Q and Qbar nodes.

Use structural description and connect nor outputs to internal signals s1 and s2. Later assign s1 and s2 to Q, Qbar.

Introduces artificial delay in driving of Q and Qbar.

Better choice is to use the driving value attribute

Part V

The IEEE Package Std_Logic 1164

- 6 Signal types in Package Std_Logic 1164
 - The resolution Function
 - Logic Functions with std_logic
- 7 Functions Defined in std_logic package 1164

9 Valued Logic

The stdlogic package uses 9 valued logic.

The basic unresolved signal type is declared as:

TYPE std_ulogic IS ('U','X','0','1','Z','W','L','H','-');

Here U is uninitialized,

X is forcing unknown, W is weak unknown,

L and H are weak 0 and 1,

Z is high impedance and - is “don't care”.

This type combines signal values and drive strengths, permitting modeling of open drain and wired or circuits. Other types are derived from this basic signal type.

Derived types

We derive the following types from the basic u_logic signal

```
TYPE std_ulogic_vector IS
```

```
    ARRAY (NATURAL RANGE<>) OF std_ulogic);
```

```
FUNCTION resolved(s:std_ulogic_vector) RETURN std_ulogic;
```

```
SUBTYPE std_logic IS resolved std_ulogic;
```

```
TYPE std_logic_vector IS
```

```
    ARRAY (NATURAL RANGE<>) OF std_logic);
```

Other Types

The IEEE package 1164 also defines the following subtypes of `std_ulogic`.

- 1 `X01` allows the values X, 0 and 1.
- 2 `X01Z` allowed the values X, 0, 1 and Z. This type is compatible with the default verilog signal type.
- 3 `UX01` allows the values U, X, 0 and 1.
- 4 `UX01Z` allows the values U, X, 0 1 and Z.

The package includes functions for conversion between various types.

The Resolution Function

This function uses the following table:

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

The resolution function receives a vector of driving values of type std_ulogic. The return is type std_ulogic!

The Resolution Function

```
FUNCTION resolved(s: std_ulogic_vector)
    RETURN std_ulogic IS
    VARIABLE result:std_ulogic:='Z'
BEGIN
    IF (s'LENGTH = 1) THEN RETURN s(s'LOW);
    ELSE
        FOR i IN s'RANGE LOOP
            result:= resolution_table(result,s(i));
        END LOOP;
    END IF;
    RETURN result;
END resolved;
```

Logic Functions with std_logic

Since signals can now acquire a multiplicity of values, we need to redefine logic functions.

This is done by overloading logic functions with new definitions when their arguments are of type std_ulogic or std_logic.

What happens when we put an inverter on a std_ulogic signal?

This is defined by the 'NOT' logic function:

NOT									
input	U	X	0	1	Z	W	L	H	-
output	U	X	1	0	X	X	1	0	X

Logic Truth TABLES

Truth tables of 2 input logic functions will now be 9x9 matrices!

AND									
	U	X	0	1	Z	W	L	H	-
U	U	U	0	U	U	U	0	U	U
X	U	X	0	X	X	X	0	X	X
0	0	0	0	0	0	0	0	0	0
1	U	X	0	1	X	X	0	1	X
Z	U	X	0	X	X	X	0	X	X
W	U	X	0	X	X	X	0	X	X
L	0	0	0	0	0	0	0	0	0
H	U	X	0	1	X	X	0	1	X
-	U	X	0	X	X	X	0	X	X

Conversion Functions

The following type conversion functions are included in package 1164:

- These include To_bit (from std_ulogic) and To_std_ulogic (from bit)
- To_bit_vector (from std_ulogic_vector and std_ulogic_vector)
- To_std_ulogic_vector (from bit_vector) and To_std_logic_vector (from bit_vector)
- To_std_logic_vector (from std_ulogic_vector) and To_std_ulogic_vector (from std_logic_vector)
- There are similar functions for inter-conversions between X01, X01Z etc. and std_logic and std_ulogic.

Edge Detection Functions

The IEEE library package 1164 includes edge detection functions for std_ulogic types. These are defined as:

```
FUNCTION rising_edge (SIGNAL s: std_ulogic)
    RETURN Boolean
```

The rising edge is detected when there is a transition from 0 or L to 1 or H.

```
FUNCTION falling_edge (SIGNAL s: std_ulogic)
    RETURN Boolean
```

The falling edge is detected when there is a transition from 1 or H to 0 or L.

Part VI

File I-O in VHDL

8 Files in VHDL

- File Declarations
- Opening and Closing Files
- Reading and writing
- Example of File usage

9 The Textio Package

Files in VHDL

To VHDL, a file is a collection of information of a type that is known to it.

- File I-O presents a special problem, because conventions for naming files and directories are different for different Operating Systems.
- We would like to insulate hardware descriptions from this variation.
- We do it by making a distinction between file names used by VHDL and the operating system dependent filename which is associated with it.

FILE Types

In VHDL, in order to use files, we use a two step procedure.

- 1 We declare a FILE **TYPE** first. This associates a File **TYPE** with the kind of objects that files of this type will contain.
- 2 We can then declare files of this FILE TYPE.
The file declaration associates a VHDL filename with a FILE TYPE and optionally, with a Physical file name and file mode (read, write or append).

Examples

TYPE datafile IS FILE OF CHARACTER;

This specifies that any file which has the type datafile will contain characters and each read will return a character while each write will accept a character to be written to the file.

Once a file type has been declared, we may declare one or more files of this type. For example,

```
FILE vfile1: datafile;
```

```
FILE vfile2: datafile IS "indata.dat"
```

```
FILE vfile3: datafile OPEN WRITE_MODE is "output.dat";
```

FILE vfile1: datafile;

This form merely associates the VHDL name vfile1 with the file TYPE datafile, which specifies that it contains characters.

FILE vfile2: datafile IS "indata.dat"

This form also associates the VHDL filename vfile2 with the Physical filename indata.dat.

FILE vfile3: datafile OPEN WRITE_MODE is "output.dat"; This form associates the vhdL filename vfile3 with the physical filename output.dat and also opens it in write mode.

Opening and Closing Files

If a file has not been opened during its declaration, it can be opened later by specific statements.

Once a file type has been declared as:

TYPE FileType IS FILE OF DataType;

it implicitly defines various procedures and functions.

```
PROCEDURE FILE_OPEN(FILE f: FileType;  
    Phys_name: IN string;  
    open_kind: IN FILE_OPEN_KIND:= READ_MODE);
```

```
PROCEDURE FILE_CLOSE(FILE f: FileType);
```

Reading from and Writing to Files

Once file types and files have been declared, various subprograms become available.

```
PROCEDURE READ(FILE f: FileType; value: OUT Data_type);  
PROCEDURE WRITE(FILE f: FileType; value: IN Data_type);  
FUNCTION ENDFILE(FILE f: FileType) RETURN Boolean;
```

Unconstrained Data Types

It is possible to declare a File Type to contained unconstrained arrays as data types. For example:

```
TYPE VectorFile IS FILE OF std_ulogiv_vector;
```

Now how do we know the amount of data which will be returned upon each read request? For this, there is an additional syntax for the read procedure:

```
PROCEDURE READ(FILE f: FileType; value: OUT Data_type  
               Length: OUT natural);
```

When we use this form, we supply an array large enough to accommodate the array in the worst case and a variable, which will receive the length of the vector actually read.

Example of File usage

```
Library IEEE;  
USE IEEE.std_logic_1164.ALL;  
ENTITY ROM_Block IS  
    GENERIC(size: NATURAL, content_file: STRING)  
    PORT(Chip_sel: IN std_logic;  
         rdbar: IN std_logic;  
         Addr: IN std_logic_vector;  
         Data: IN std_logic_vector);  
END ENTITY ROM_Block;
```

ROM Initialization

```
ARCHITECTURE From_File OF ROM_Block IS
    SUBTYPE Word IS
        std_logic_vector(Data'Length-1 DOWNT0 0);
    TYPE Mem_Array IS
        ARRAY(NATURAL RANGE 0 TO 2**size -1) of Word;
    VARIABLE Mem_Contents: Mem_Array;
    VARIABLE Index: Natural;

    ...
    TYPE RomData_File IS FILE of WORD;
    FILE Rom_Contents : RomData_FILE
        OPEN Read_Mode IS content_file;

    ...
```

ROM Initialization

```
BEGIN
  Filling: Process IS
  BEGIN
    Index := 0;
    WHILE NOT EndFile(ROM_Contents) LOOP
      READ(ROM_Contents, Mem_Contents(Index);
      Index:= Index+1;
    END LOOP;
  WAIT;
  END PROCESS Filling;
  ...      - - process to handle rdbar
END ARCHITECTURE From_File;
```


The Textio Package

This package defines various TYPEs and provides many procedures for handling text.

```
TYPE TEXT IS FILE OF STRING;
```

```
TYPE LINE IS ACCESS STRING;
```

```
FILE INPUT: TEXT OPEN READ_MODE IS "std_input"
```

```
FILE OUTPUT: TEXT OPEN WRITE_MODE IS "std_output"
```

```
PROCEDURE READLINE(FILE f: TEXT; L: INOUT LINE)
```

Reading and Writing Text

Text reading and writing is a two step procedure. For writing, you first compose a line and then write it to a file. For reading, you read a line and then extract values from it.

Several overloaded functions all carrying the names READ or WRITE are provided for this. For example:

```
PROCEDURE READ (L: InOut LINE; value: OUT BIT);  
PROCEDURE READ (L: InOut LINE; value: OUT  
BIT_VECTOR);  
PROCEDURE READ (L: InOut LINE; value: OUT Integer);  
PROCEDURE READ (L: InOut LINE; value: OUT BIT);  
etc.
```

Similarly, there are many WRITE functions.

Part VII

An Example Design

- 10 A magnitude comparator
 - First Level Description
 - Constructing the Byte Comparator
 - Structural Description of Bit Comparator

A Magnitude Comparator

- The example used in this section has been described in the book: “VHDL: Analysis and Modeling of Digital Systems” by Zainalabedin Navabi (McGraw Hill).
- However the treatment in this tutorial is different.
- We illustrate top down design using this example.

A magnitude comparator

- We want to design a circuit to compare the magnitude of two binary numbers.
- We shall illustrate the design by a comparator for byte wide numbers.
- However, the design should be stackable, so that wider numbers can be compared.
- The input to the system are the two numbers and stacking inputs, `gt_in`, `eq_in` and `lt_in`.
- The outputs are the result of comparison: `gt_out`, `eq_out` and `lt_out`.
- The stacking inputs and outputs use “one hot” coding: exactly one of the conditions `gt`, `eq` or `lt` is TRUE at a given time.

First level description

```
Library IEEE;  
USE IEEE.std_logic_1164.ALL;  
TYPE Byte IS Array (7 DownTo 0) OF std_ulogic;  
Entity Byte_Compar is  
    Port(a, b: IN BYTE;  
          gt_in, eq_in, lt_in: IN std_ulogic;  
          gt_out, eq_out, lt_out: OUT std_ulogic);  
End Entity Byte_Compar;
```

Architecture of Byte Comparator

Architecture first Of Byte_Compar is

Variable val1, val2: Integer:= 0;

BEGIN

P1: PROCESS(a, b, gt_in, eq_in, lt_in)

BEGIN

val1 := ByteVal(a);

val2 := ByteVal(b);

IF (val1 > val2) THEN

gt_out <= '1'; eq_out <= '0'; lt_out <= '0';

ELSIF (val1 < val2) THEN

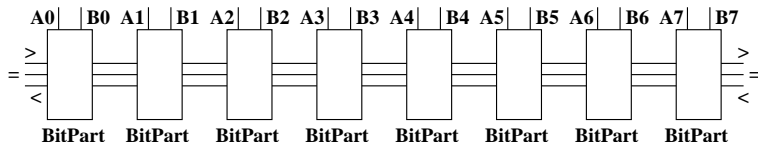
gt_out <= '0'; eq_out <= '0'; lt_out <= '1';

ELSE gt_out <= gt_in; eq_out <= eq_in; lt_out <= lt_in;

END IF;

Decomposition of Byte Comparator

The byte comparator is difficult to design directly.
We can break up the design into bit comparators
with cascading inputs `gt_in`, `eq_in` and `lt_in`;
and cascading outputs `gt_out`, `eq_out` and `lt_out`.



Notice that the most significant bit is compared closest to the output.

Composing the Byte comparator

Architecture compose of Byte_Compar IS

```
    COMPONENT BitPart IS
```

```
        Port(a, b: IN std_ulogic;
```

```
              gt_in, eq_in, lt_in: IN std_ulogic;
```

```
              gt_out, eq_out, lt_out: OUT std_ulogic);
```

```
    END COMPONENT BitPart;
```

```
    FOR ALL: BitPart
```

```
        USE ENTITY Bit_Compar(behave);
```

```
    TYPE Connect IS ARRAY (1 TO 3, 0 TO 6) OF std_ulogic);
```

```
    Signal Cascade: Connect;
```

Composing the Byte comparator

```
BEGIN
  FOR I in 0 TO 7 GENERATE
    First: IF I = 0 GENERATE
      COMPONENT BitPart
        PORTMAP
          (gt_in, eq_in, lt_in,
           a(I), b(I),
           Connect(1, I), Connect(2, I), Connect(3, I));
      END GENERATE;
```

Composing the Byte comparator

```
Last: IF I = 7 GENERATE
      COMPONENT BitPart
        PORTMAP
          (Connect(1, I-1), Connect(2,I-1), Connect(3,I-1));
          a(I), b(I),
          gt_out, eq_out, lt_out)
      END GENERATE;
```

Composing the Byte comparator

```
Mid: IF (I > 0) AND (I < 7) GENERATE
      COMPONENT BitPart
        PORTMAP
          (Connect(1, I-1), Connect(2, I-1), Connect(3, I-1));
          a(I), b(I),
          Connect(1, I), Connect(2, I), Connect(3, I));
      END GENERATE;
END GENERATE;
END Architecture Compose;
```

The bit comparator

Once we have decomposed the byte comparator as above, we need to design the bit comparator.

- The bit comparators receive a pair of bits to compare.
- If $A > B$, i.e. $A=1$ and $B=0$; it makes the output `gt_out` TRUE and makes the other outputs FALSE.
- If $A < B$, i.e. $A=0$ and $B=1$; it makes the output `lt_out` TRUE and makes the other outputs FALSE.
- IF A and B are equal, it copies its cascading inputs (`gt_in`, `eq_in`, `lt_in`) to its outputs (`gt_out`, `eq_out`, `lt_out`);

The bit comparator

```
Library IEEE;  
USE IEEE.std_logic_1164.ALL;
```

```
Entity Bit_Compar is  
    Port(a, b: IN std_ulogic;  
          gt_in, eq_in, lt_in: IN std_ulogic;  
          gt_out, eq_out, lt_out: OUT std_ulogic);  
End Entity Bit_Compar;
```

Behavioural Architecture of Bit Comparator

Architecture behave Of Bit_Compar is

BEGIN

P1: PROCESS(a, b, gt_in, eq_in, lt_in)

BEGIN

IF (a = '1' AND b = '0') THEN

gt_out <= '1'; eq_out <= '0'; lt_out <= '0';

ELSIF (a = '0' AND b = '1') THEN

gt_out <= '0'; eq_out <= '0'; lt_out <= '1';

ELSE gt_out <= gt_in; eq_out <= eq_in; lt_out <= lt_in;

END IF;

END PROCESS P1;

END Architecture behave;

Structural Description of Bit Comparator

We can write Karnaugh Maps for the three outputs easily:

gt_out				
ab → gt_in ↓	00	01	11	10
0				✓
1	✓		✓	✓

lt_out				
ab → lt_in ↓	00	01	11	10
0		✓		
1	✓	✓	✓	

eq_out				
ab → eq_in ↓	00	01	11	10
0				
1	✓		✓	

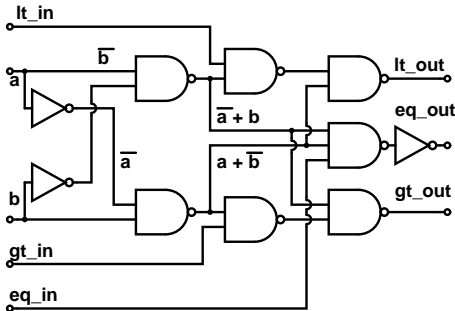
This gives:

$$gt_out = a \cdot \bar{b} + gt_in \cdot (a + \bar{b})$$

$$lt_out = \bar{a} \cdot b + lt_in \cdot (\bar{a} + b)$$

$$eq_out = eq_in \cdot (a \cdot b + \bar{a} \cdot \bar{b})$$

Final Design of bit comparator



- This design can be described structurally in terms of basic gates.
- The design uses only inverting gates. It can be implemented directly on a chip.

Structural Description of Bit Comparator

Architecture struct Of Bit_Compar is

Component Inv IS

PORT(In1: IN std_ulogic; op1: OUT std_ulogic);

END COMPONENT Inv;

FOR ALL: Inv USE ENTITY Inverter(behav);

Component Nand2 IS

PORT(In1, In2: IN std_ulogic; op1: OUT std_ulogic);

END COMPONENT Nand2;

FOR ALL: Nand2 USE ENTITY Nand2(behav);

Component Nand3 IS

PORT(In1, In2, In3: IN std_ulogic; op1: OUT std_ulogic);

END COMPONENT Nand3;

FOR ALL: Nand3 USE ENTITY Nand3(behav);

Structural Architecture of Bit Comparator

```
SIGNAL Abar, Bbar, AplusBbar, BplusAbar: std_ulogic;  
SIGNAL s1, s2, Eqbar: std_ulogic;  
BEGIN  
Inv1:    Inv PORTMAP(A, Abar);  
Inv2:    Inv PORTMAP(B, Bbar);  
N1:      Nand2 PORTMAP(A, Bbar, BplusAbar);  
N2:      Nand2 PORTMAP(B, Abar, AplusBbar);  
N3:      Nand2 PORTMAP(lt_in, BplusAbar, s1);  
N4:      Nand2 PORTMAP(gt_in, AplusBbar, s2);  
N5:      Nand2 PORTMAP(s1, AplusBbar, lt_out);  
N6:      Nand2 PORTMAP(s2, BplusAbar, gt_out);  
N7:      Nand3 PORTMAP(AplusBbar, BplusAbar, Eq_in, Eqbar);  
Inv3:    Inv PORTMAP(Eqbar, Eq_out);
```

Inline configuration

The configuration of a component can be declared “inline” in an architecture.

Architecture compose of Byte_Compar IS

```
    COMPONENT BitPart IS
```

```
        Port(a, b: IN std_ulogic;
```

```
            gt_in, eq_in, lt_in: IN std_ulogic;
```

```
            gt_out, eq_out, lt_out: OUT std_ulogic);
```

```
    END COMPONENT BitPart;
```

```
    FOR ALL: BitPart
```

```
        USE ENTITY Bit_Compar(behave);
```

```
    TYPE Connect IS ARRAY (1 TO 3, 0 TO 6) OF std_ulogic);
```

```
    Signal Cascade: Connect;
```

All components of type BitPart have been configured to use the

Standalone configuration

- In the example given, all components of type BitPart were configured to use the entity Bit_Compar with architecture behave.
- This was specified "inline" in the architecture declarative part.
- We can write a separate configuration description outside the architecture using the configuration.

Stand alone configuration

The syntax of a standalone configuration is:

```
CONFIGURATION configname OF entityname IS
    FOR architecture_name
        FOR instance_name | OTHERS | ALL : component_name
            USE ENTITY sub_entity_name(sub_architecture_name)
            ...
        END FOR;
    END FOR;
END [CONFIGURATION] [configname];
```

Hierarchical configuration

- The architecture being configured may contains components which are bound to architectures containing other components.
- This requires hierarchical configuration.
- Instead of binding component instances to entity-architecture pairs directly, we bind these to other configurations.
- These other configurations associate the component with an entity-architecture pair and cofigure the lower level components.

Hierarchical configuration

The syntax used for hierarchical configuration is:

```
CONFIGURATION configname OF entityname IS
    FOR architecture_name
        FOR instance_name | OTHERS | ALL : component_name
            USE CONFIGURATION subconfig_name;
        ...
    END FOR;
END FOR;
END [CONFIGURATION] [configname];
```

Subconfig_name will associate the component with an entity-architecture pair and will configure lower level components in the hierarchy.

Hierarchy in a single configuration

The hierarchy can be described through nested FORs in a single configuration description.

CONFIGURATION single OF Byte_compar IS

FOR compose – architecture name

FOR ALL: BitPart

USE ENTITY WORK.Bit_Compar(struct);

FOR struct – architecture of Bit_Compar

FOR ALL: Nand2 USE ENTITY ...